Modular translation validation of a full-sized synchronous compiler using off-the-shelf verification tools (abstract)

van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier, Loic Besnard, Paul Le Guernic

Modular translation validation of a full-sized synchronous compiler using off-the-shelf verification tools (abstract)

Van-Chan Ngo Jean-Pierre Talpin Thierry Gautier Loïc Besnard Paul Le Guernic
INRIA Rennes, France
first.lastname@irisa.fr

ABSTRACT

The aim of this presentation is to demonstrate a scalable, modular, refinable methodology to design, assess and improve the trustability of an existing (20 years old), large (500k lines of C), open source (Eclipse/Polarsys IWG project POP) code generation suite using off-the-shelf, open-source, SAT/SMT verification tools (Yices), by adapting and optimizing the translation validation principle introduced by Pnueli et al. in 1998. This methodology results from the ANR project VERISYNC, in which we aimed at revisiting Pnueli's seminal work on translation validation using off-the-shelf, up-to-date, verification technology. In face of the enormous task at hand, the verification of a compiler infrastructure comprising around 500 000 lines of C code, we devised to narrow down and isolate the problem to the very data-structures manipulated by the infrastructure at the successive steps of code generation, in order to both optimize the whole verification process and make the implementation of a working prototype at all doable. Our presentation outlines the successive steps of this endeavour, from clock synthesis, static scheduling to target code production.

1. INTRODUCTION

Synchronous programming languages like Signal, Lustre and Esterel propose a formal semantic framework to give high-level specification of safety-critical software in automotive and avionics systems [2, 14, 15, 17]. Safety-critical systems are those systems whose failure could result in loss of life, or damage to the environment. They need to be validated to ensure that their specified safety properties are implemented correctly. Software validation is traditionally done by using testing techniques which, in the case of safety-critical systems, is not sufficient [19]. Since synchronous languages are based on formal semantic models, they provide much higher level of abstraction, expressivity, and clarity at source level rather than once compiled into C code. That makes the application of formal methods much simpler to enforce safety properties. However, a synchronous compiler is still a large and complex program which often consists of hundreds of thousands lines of code, divided into numerous packages. Moreover, compiler modules often interact in sophisticated ways, and the design and implementation of a compiler are substantial engineering tasks. The compilation process involves many analyzes, program transformations and optimizations, some may introduce additional information or constrain the compiled program, some may refine its meaning and specialize its behavior to meet a specific safety or optimization goal, and all these compile-time decisions should additionally be traced.

2. STATE OF THE ART

Proving the correctness of a compiler can be based on the examination of the developed compiler's source code itself, meaning that a qualification process applies on the development of the compiler, the source of the compiler, and/or the compiler's output. Qualifying a compiler is rare because of the tremendous administrative effort involved. Qualification amounts to demonstrate compliance with all recommendations and objectives specified in the certification standards for safety-critical softwares: Do-178C and its European equivalent Ed-12 [31]. Although Do-178 has been successful in industry, the cost of complying with it is significant: the activities on verification it incurs may well cost seven times more than the development effort needed [30]. A more traditional method is therefore to solely inspect or formally verify the compiler's output. This task requires less unitary effort, but has to be repeated every time target code is generated. For instance, Astrée [1] [3] is a special-purpose static program analyzer based on abstract interpretation to verify the absence of *run time errors* in the C code generated from Scade programs. One last resort is to formally verify the correctness of the compiler itself. *Certifying compilation* [20] attests that the generated object code satisfies the properties established on the source program by generating concrete evidences along the compilation into object code. Systematic compiler verification techniques use formal methods. For the purpose of compiler verification, there are two approaches to prove the software correctness.

Formal compiler verification consists of specifying the behavior of the compiler in a formal specification language and build a proof that the compiler satisfies behavioral equivalence or refinement. Formal verification can be done through many approaches. One such approach is *deductive* verification. It consists of providing deductive proofs that a system behaves in a certain way that is described in the specification, with the aid of either interactive theorem provers (such as Hol [13], Isabelle [16], or Coq [9]), or an automated theorem prover. Another approach is *model checking* [6, 29]. It involves building an abstract model of the system and ensure it complies with specified requirements by exploring all its accessible states. Requirements are represented in *temporal logics*, such as *Linear Temporal Logic* (Ltl) or *Computational Tree Logic* (Ctl) and verification produces a confirmation that the system model conforms to requirements or a counterexample that can be used to locate and eliminate an error. Some techniques can be used to deal with the *state explosion* problem including *abstract interpretation*, *symbolic simulation* and *abstract refinement* [10]. A variant of model checking, *Bounded model checking* (Bmc) [5], encodes the fact that potential executions of the system model do not conform to the specification in incremental fashion as propositional satisfiability formulas. The bounded number of evaluation steps is increased as long as the resulting propositional formula is satisfiable. Then a concrete counterexample can be extracted as a trace of system states leading to an error state in the system model. *Inductive* reasoning can also be used to prove that a system conforms its specification. Advances in *Satisfiability Modulo Theories* (Smt) have been useful for checking systems inductively. With SMT solvers, systems can be modelled efficiently, require fewer limitations to represent specifications and meet significant performances.

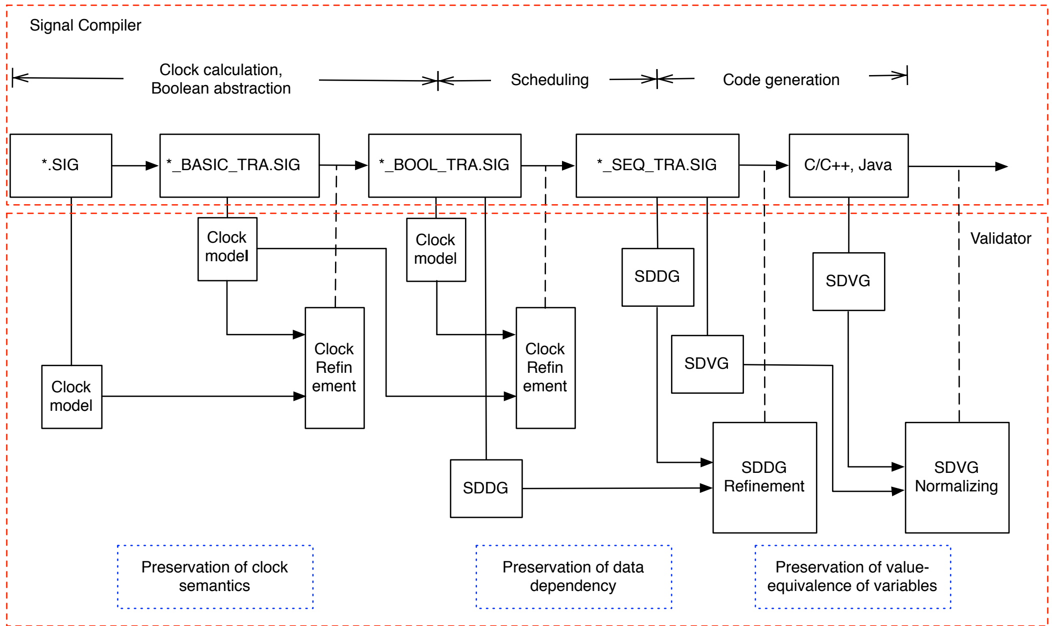


Figure 1 The translation validation for the SIGNAL compiler

Translation validation was introduced by Pnueli et al. in [27] as an approach to verify the correctness of translators (compilers, code generators). The main idea of translation validation is that instead of proving the correctness of a compiler, each of its individual translations (e.g., run of the code generator) is followed by a validation phase to check that the target program correctly implements the source specification. A translation validator consists of two objects:

- *The Model builder* is a simple module that formally represents the semantics of the source and target programs of the translator (e.g., labeled transition system, first-order logic formula).
- *The Analyzer* formalizes correctness as a refinement relation between the models of the source and target of the validator. The analyzer provides an automated proof method on the existence of the refinement between the formal models. If the analyzer succeeds, a *proof script* is created. If it doesn't, it generates a *counter example*, which can be decompiled to help spot the error.

Translation validation does not modify or instrument the compiler. It treats it as a "black box" (as long as there is no error in it). It only considers the input program and its compiled result. Hence, it is not affected by updates and modifications to the compiler, as long as its data-structures remain the same. In general, the validator is much simpler and smaller than the compiler itself. Thus, the proof of correctness of the validator takes less effort than that of the compiler. Plus, verification is fully automated and scales to large programs.

3. SCALABLE TRANSLATION VALIDATION

Our approach is to scale translation validation not only in a modular fashion, by decomposing the problem into the successive transformations performed by the compiler on the intermediate representation of a program [22], but by narrowing it further to the actual data-structure that are being used to represent the transformation problem and the actual algorithmic operations performed on it. In all cases, we show that translation validation is amenable to simple SAT/SMT verification (the semantic inclusion of one data-structure into another) instead of the more general problem of simulation-based conformance-checking of the transformed program w.r.t. the input specification [23–25]. In the case of the synchronous data-flow language SIGNAL, a 500k-lines big code generation infrastructure, the compilation process can be divided into three phases depicted in Figure 1 (top row). A source Signal program is the synchronous composition of discrete equations on signals, e.g. $x := y + 1 \mid y = x\$1$ defines x by $y + 1$ at all times and y by the value of x delayed by 1 evaluation tick: its previous value. Its compilation may be seen as a sequence of morphisms that refine and rewrite the source specification with information gained from analysis. C or JAVA code production is performed on ultimately transformed program, e.g., $y = x;\ x := y + 1$.

- *Clock calculation*. This stage determines the clock of all signals in the program and defines a Boolean abstraction of the program. The clock of a signal defines when the value of the signal shall be evaluated.
- *Static scheduling*. Based on the clock information and the Boolean abstraction obtained at the first stage, the compiler constructs a *Conditional Dependency Graph* (CDG), which represents the schedule of signals' evaluations.
- *Code generation*. Sequential C or Java code is directly generated from the structure of the clocked and scheduled Signal program.

A proof of semantic-preservation can be decomposed into the preservation of clock semantics at the *clock calculation* phase and that of data dependencies at the *static scheduling* phase. Value-equivalence of variables can be then checked at the *code generation* phase. Figure 1 shows the integration of this verification framework into the compilation process of the SIGNAL compiler. For each phase, the validator takes the source program and its compiled counterpart, and constructs the corresponding formal model of the program. Then, it checks the existence of the refinement relation to prove semantic-preservation.

3.1 Preservation of clock models [24]

The first verification stage focuses on proving that all clock relations associated with *signals* in the source and transformed program are equivalent. A *clock model* is a first-order logic formula that characterizes the presence/absence status of all signals at all times in a SIGNAL program. Given two clock models, a *clock refinement* relation is defined to express the semantic preservation of clock semantics. The existence of a clock refinement is defined as a satisfiability problem which is automatically and efficiently proved by an SMT solver.

Example Consider the Signal program `DEC` which counts through the output `N` from the value of input `FB` to 1, `ZN := N$1 init 1` defined `ZN` as the previous value of `N`, `N := FB default (ZN-1)` assigns `FB` to `N` when the input `FB` is present, and `ZN-1` otherwise, `FB ^= when (ZN<=1)` synchronizes `FB` to the condition `(ZN<=1)`. The clock model of the source program is:

$$
\begin{aligned}
&\Phi(\texttt{DEC}) = \\
&\quad (\widehat{FB} \Leftrightarrow \widehat{ZN_1} \wedge \widetilde{ZN_1}) \\
&\wedge (\widehat{ZN_1} \Leftrightarrow \widehat{v^1_{<=}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN_1} \Rightarrow (\widetilde{ZN_1} = \widetilde{v^1_{<=}})) \\
&\wedge (\widehat{ZN} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN} \Rightarrow (\widetilde{ZN} = m.N \wedge m.N' = \widetilde{N})) \wedge (m.N_0 = 1) \\
&\wedge (\widehat{N} \Leftrightarrow \widehat{FB} \vee \widehat{ZN_2}) \wedge (\widehat{N} \Rightarrow ((\widehat{FB} \wedge \widetilde{N} = \widetilde{FB}) \vee (\neg\widehat{FB} \wedge \widetilde{N} = \widetilde{ZN_2}))) \\
&\wedge (\widehat{ZN_2} \Leftrightarrow \widehat{v^1_{-}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN_2} \Rightarrow (\widetilde{ZN_2} = \widetilde{v^1_{-}}))
\end{aligned}
$$

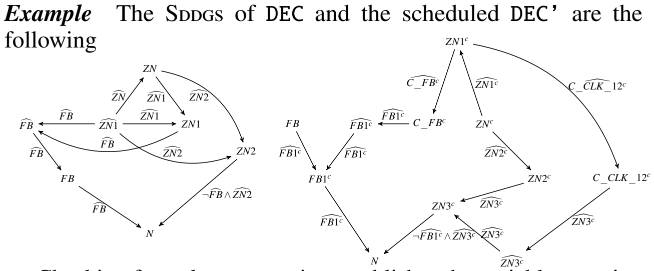
Terms $\hat{x}$, resp. $\tilde{x}$, represent the clock resp. value of a signal x. The model of the transformed program `DEC'` is twice as large. Checking it a correct refinement amounts to establish a variable mapping $\widehat{X_{\texttt{DEC}}} \backslash \widehat{X_{IO}} = \alpha(\widehat{X_{\texttt{DEC'}}} \backslash \widehat{X_{IO}})$ between `DEC` and `DEC'` and delegate the checking validity of the formula $(\Phi(\texttt{DEC}') \wedge \widehat{X_{\texttt{DEC}}} \backslash \widehat{X_{IO}} = \alpha(\widehat{X_{\texttt{DEC\_BASIC\_TRA}}} \backslash \widehat{X_{IO}}) \Rightarrow \Phi(\texttt{DEC}))$, named φ, to the SMT solver under the logical context defined by the variable mapping and the following assertions:

$$(\widetilde{ZN} = \widetilde{ZN^c} \wedge 1 = 1) \Rightarrow \left((\widetilde{v^1_{<=}} = \widetilde{v^{1c}_{<=}}) \wedge (\widetilde{v^1_{-}} = \widetilde{v^{1c}_{-}}) \right)$$

3.2 Preservation of data dependency [25]

The goal of this stage is to prove that the existence of a data dependency between two signals in the source specification is a property preserved by the target program which, in addition, makes a sequential schedule of computations explicit. Along the way, the validator further checks the target program *deadlock-free*. Data dependencies among signals are represented by a *Synchronous Data-flow Dependency Graph* (SDDG). An SDDG is a labeled directed graph in which node are signals and clocks and edges represent dependencies between nodes. Edge are labeled by clocks. An edge clock tells when the dependency is effective: when its clock is present. The correctness of a schedule is formalized as a *dependency refinement* relation between the source and target SDDGS. It is implemented by SMT-checking the existence of the refinement relation.

Example The SDDGS of `DEC` and the scheduled `DEC'` are the following



Checking formula construction establishes the variable mapping much like for clock models. Next, it amounts to finding all cycles and dependency paths from `FB` to `N` to generate the formulas for checking the dependency refinement, the validity of which is delegated to the YICES solver.

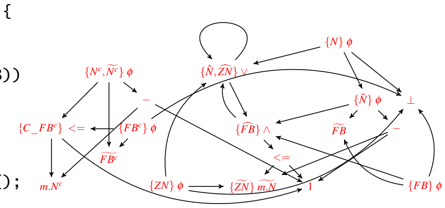
$$\widehat{FB} \Rightarrow (\widehat{FB1^c} \wedge \widehat{FB1^c})$$

3.3 Value-equivalence of variables [23]

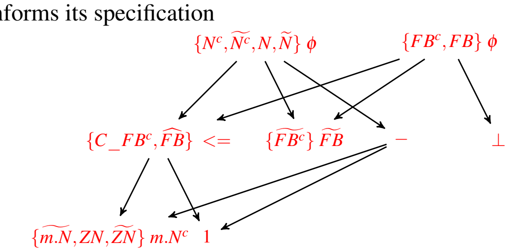
This stage focuses on proving that every output signal in the source program and the corresponding variable in the generated C program are assigned the same values at all times. The defining equation of a signals and its C translation are represented by a shared value-graphs, called *Synchronous Data-flow Value-Graph* (SDVG). To prove that a signal and a variable have the same value with an SDVG, we just need to check that they are represented by the same sub-graph, meaning that they point to the same graph node.

Example The code generated from `DEC` consists of the following step function (left). The value graphs of `DEC` and `DEC_step` are constructed and composed (right).

```
logical DEC_step() {
 C_FB = N <= 1;
 if (C_FB)
  if (!r_DEC_FB(&FB))
    return FALSE;
 if (C_FB) N = FB;
   else N = N - 1;
 w_DEC_N(N);
 DEC_step_finalize();
 return TRUE;
}
```



The shared SDVG are then normalized by employing a series of rewriting rules to merge nodes refering to the same value. This yields the following graph, proving that the generated step function conforms its specification



4. RELATED WORKS

The notion of translation validation was introduced in [27, 28] by Pnueli et al., using a symbolic model of *Synchronous Transition Systems* (STS) to represent both source and target programs. An STS is a set of logic formulas which describes the functional and temporal constraints of the whole program and its generated C code. BDDS [4] implement the symbolic STS models. Our approach improves over standard translation validation by explicitly capturing the clock semantics in the model, which also results in much smaller models. Another related work is the static analysis of SIGNAL programs for efficient code generation [12], where linear relations among clocks and values are represented by first-order logic formulas with the help of numerical interval abstraction techniques. The objective is to make generated code more efficient by detecting and removing the dead-code segments (e.g., segment of code to compute a data-flow which is always absent). They determine the existence of empty clocks, mutual exclusion of two or more clocks, or clock inclusions, by reasoning on the formal model using an SMT solver. Related works have also adopted the translation validation approach in verification of transformations, and optimizations. In [21], the translation validation is used to verify several common optimizations such as common subexpression elimination, register allocation, and loop inversion. The validator checks the existence of a simulation relation between two programs. Leroy et al. [7, 18] used this technique to develop the COMPCERT high-assurance C compiler. The programs before and after the transformations and optimizations of the compiler are represented in a common intermediate form, then the preservation of semantics is checked by using symbolic execution in the proof assistant COQ. Tristan et al. [32] recently proposed a framework for translation validation of LLVM optimizer. For a function and its optimized counterpart, they compute a shared value-graph. The graph is *normalized*. If the outputs of two functions are represented by the same sub-graph, they can safely conclude that two functions are equivalent. We believe that our approach is more modular and efficient in both design time, space and time than these based on proof automation and simulation relations: by reducing each of the translation steps to these of the very data-structures subject to refinements (the clock hierarchy, the data-flow graph, the value graphs) considerably reduces the size of the refinement or simulation problem to solve, and that using off-the-shelf verifiers like Z3, Yices, SMTLib guarantees both speed and correctness.

5. CONCLUSION

We have presented a technique based on SMT solving to prove the preservation of clock semantics during the compilation of a synchronous data-flow compiler. Our approach focuses on the transformations performed by the compiler using the simplest structures to represent them: SAT/SMT formulas represent the refinement of clock models and the reinforcement of data-flow graphs, value graphs are used to represent the production of target code patterns from and a specification's syntax tree. This reduces the whole process of proving a refinement relation between the source specification and the generated code to a couple of SMT SAT-checking on formulas of minimal size, and to a symbolic rewriting on a reduced graph to check value equivalence. Our validator does not modify or instrument the compiler. It treats it as a "black box" (as long as there is no error in it). It only considers an input program and its transformed result. Hence, it is not affected by an update an modification made to this or that compilation stage, as long as its principle and data-structure remains the same. The validator is much simpler and smaller than the compiler itself. Proving its correctness (the model builder, the verifier) would take a lot less effort than for the compiler as well. Verification is fully automated and scales to large programs very well by employing state-of-the-art verification tools and by minimizing the representation of the problem to solve. For that purpose, we represent the desired program semantics using a scalable abstraction and we use efficient SMT libraries [11] to achieve the expected goals: traceability and formal evidence. We believe that this approach provides a, both technically and economically, attractive alternative to developing a certified compiler. The individual modules designed in the context of this project are being integrated in the open-source environment of the Eclipse project POP with the Polarsys Industry Working Group [26].

REFERENCES

[1] Astrée, http://www.astree.ens.fr.

[2] G. Berry. The foundations of Esterel. In Proof, Language and Interaction: Essay in Honor of Robin Milner, MIT Press, 2000.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In ACM Conference on Programming Language Design and Implementation, 2003.

[4] R. Bryant, Graph-based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, 35(8), 1986.

[5] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. Internationa Conference on Formal Methods in System Design, 19, 2001.

[6] E.M. Clarke, O. Grumberg, and D.A. Peled. Automatic verification of finite-state concurrent systems using temporal logic specifications. Model Checking, The MIT Press, Cambridge, MA, 2000.

[7] Inria, The CompCert Project. http://compcert.inria.fr.

[8] K. Cooper and L. Torczon. Engineering a Compiler, Second Edition. Morgan Kaufmann, 2011.

[9] Coq-Inria. Coq proof assistant. http://coq.inria.fr.

[10] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. Automated Software Engineering, 6, 1999.

[11] The Yices Sat-solver. http://yices.csl.ri.com.

[12] A. Gamatié, and L. Gonnord, Static Analysis of Synchronous Programs in Signal for Efficient Design of Multi-Clocked Embedded Systems. In ACM Conference on Languages, Compilers, Tools and Theory for Embedded Systems, April 2011.

[13] M.J.C. Gordon and T.F. Melham. Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, 1993.

[14] P. Le Guernic, M. Le Borgne, T .Gautier, and C. Le Maire. Programming real-time applications with Signal. Proceedings of the IEEE Special Issue, 1991.

[15] N. Halbwachs. A synchronous language at work: the story of lustre. ACM-IEEE International Conference on Formal Methods and Models for Codesign, 2005.

[16] The Isabelle proof assistant. http://www.cl.cam.ac.uk/research/hvg/Isabelle.

[17] D. Jackson. A direct path to dependable software. Communications of the ACM, pages 52(4):78–88, 2009.

[18] X. Leroy, Formal Certification of a Compiler Back-end, or Programming a Compiler with a Proof Assistant. ACM Symposium Principles of Programming Languages, 2006.

[19] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or formal verification: Do-178c alternatives and industrial experience. IEEE Software, 2013.

[20] G. C. Necula. Proof-carrying code. ACM Symposium Principles of Programming Languages, 1997.

[21] G.C. Necula, Translation Validation for an Optimizing Compiler. ACM SIGPLAN Conference on Programming Language Design and Implementation, May 2000.

[22] V. C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard. Formal Verification of Compiler Transformations on Polychronous Equations. Integrated Formal Methods, LNCS 7321, 2012.

[23] V. C. Ngo, J.-P. Talpin, T. Gautier. Translation Validation for Synchronous Data-flow Specification in the SIGNAL Compiler. International Conference on Formal Techniques for Distributed Objects, Components and Systems. IFIP, 2015.

[24] V. C. Ngo, J.-P. Talpin, T. Gautier. Translation validation for clock transformations in a synchronous compiler. International Conference on Fundamental Approaches to Software Engineering. Springer, 2015.

[25] V. C. Ngo, J.-P. Talpin, T. Gautier. Efficient deadlock detection for polychronous data-flow specifications. Electronic System Level Synthesis Conference. IEEE, 2014.

[26] Polarsys project POP, Polychrony On Polarsys. https://www.polarsys.org/projects/polarsys.pop.

[27] A. Pnueli, M. Siegel, and E. Singerman, Translation Validation. International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 1998.

[28] A. Pnueli, O. Shtrichman, and M. Siegel, Translation Validation: From Signal to C. Correct System Design Recent Insights and Advances, LNCS 1710, Springer, 2000.

[29] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in Cesar. ACM Symposium on Principles of Programming Languages, 1981.

[30] J. Rushby. New challenges in certification for aircraft software. ACM Conference on Embedded Software, 2011.

[31] RTCA DO-178. Software considerations in airborne systems and equipment certification. RTCA and EUROCAE, 2011.

[32] J-B. Tristan, P. Govereau, and G. Morrisett, Evaluating Value-graph Translation Validation for LLVM. ACM Conference on Programming and Language Design Implementation, June 2011.